

Algoritmos y Estructuras de Datos II

Segundo parcial – Sábado 24 de junio de 2023

#Orden	Nro. Libreta	Apellido y Nombre	Ej1	Ej2	Ej3	Nota Final

- Es posible tener una hoja (2 carillas), escrita a mano, con los apuntes que se deseen + el apunte de Módulos Básicos.
- Cada ejercicio debe entregarse **en hojas separadas**.
- Incluir en cada hoja el número de orden asignado, número de libreta, número de hoja, apellido y nombre.
- Cada ejercicio se calificará con **Perfecto**, **Aprobado**, **Regular**, o **Insuficiente**.
- El parcial estará aprobado con las siguientes notas por ejercicio : $< R, R, A >$, $< *, A, A >$, $< A, *, A >$ o $< A, A, R >$.

Donde se menciona a **secuencia** en las definiciones de los siguientes ejercicios se debe usar alguno de los módulos vistos que se explican con el TAD SECUENCIA de acuerdo al apunte de Módulos Básicos o definir un módulo nuevo.

Ej. 1. Sorting

Se quiere obtener los resultados de una competencia de escalada. Así, el resultado de cada competidor se registrará como una **tupla(nombre: string, puntos: nat, intentos:nat)**, donde *nombre* es el nombre del competidor, *puntos* es la cantidad de puntos acumulados e *intentos* es la cantidad de veces que el competidor intentó escalar una ruta. El puntaje obtenido variará según la competencia, por lo que no existe un máximo establecido, y cada competidor puede realizar la cantidad de intentos que quiera.

Se requiere crear un algoritmo para hacer un *ranking* de los competidores:

TABLAKPOSICIONES(in a: secuencia(tupla(nombre:string, puntos:nat, intentos:nat)), in k:nat) \rightarrow res:secuencia(string)

Por ejemplo, dada la siguiente entrada:

R=[(Marcos, 25, 5), (Irene, 50, 2), (Pedro, 50, 2), (Lucas, 10, 10), (Juana, 25, 2)]

TablaKPosiciones(R, 3) = [Irene, Pedro, Juana]

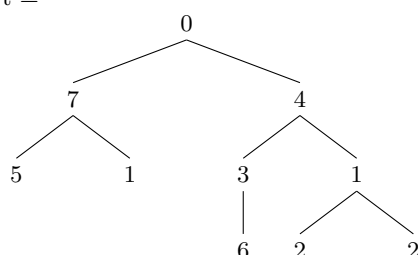
Al solicitar el *ranking* de tres (3) posiciones Irene y Pedro quedan primeros, ya que tienen el mayor puntaje, pero Irene aparece antes en el arreglo original. Después vienen Marcos y Juana, que tienen el mismo puntaje, pero Juana tiene menos intentos y queda antes, por lo que Marcos se queda fuera de los mejores tres.

Se pide escribir un algoritmo que tome el arreglo de resultados de una competencia de escalada, de tamaño n , y arme la tabla de posiciones con el top k de los mejores competidores TABLADEKPOSICIONES. La tabla de posiciones es una lista donde los competidores aparecen ordenados decrecientemente según el puntaje obtenido. En caso de empate, se desempata según la cantidad de intentos realizados de forma creciente. El algoritmo debe ser estable y tener complejidad $O(n + k \log(n))$ de peor caso, lo que debe estar justificado en forma detallada.

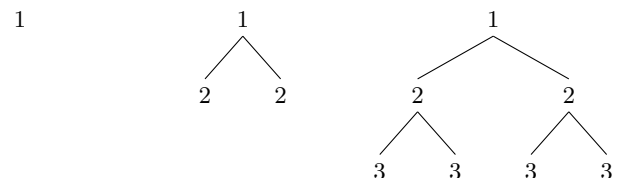
Ej. 2. Divide & Conquer

Los árboles binarios podados (ABP) son árboles binarios balanceados en altura. Es decir, las alturas de los subárboles izquierdo y derecho de cada nodo difieren en a lo sumo una unidad (como en los AVL). En los ABP no hay ninguna relación de orden entre un nodo padre y sus hijos (como en los ABB). Un árbol pirámide incremental es un árbol completo tal que todos los nodos del nivel i -ésimo tienen valor i . Estos árboles cuentan al menos con las operaciones RAÍZ, IZQUIERDA, DERECHA y NIL?, como las usadas habitualmente.

t =



Por ejemplo, abajo se representan tres árboles pirámide incrementales de tamaño 1, 2 y 3, respectivamente:



- a) Se pide dar un algoritmo que use *Divide & Conquer* para resolver el problema de encontrar el tamaño del SUBÁRBOLPIRÁMIDEINCREMENTAL (SAPI) más grande dentro de un ABP de naturales cuya complejidad sea $O(n \log(n))$ en peor caso, lo que deberá estar justificado en forma detallada

SAPI(in a: abp(nat)) \rightarrow res:nat

Por ejemplo, dado el árbol de la izquierda, SAPI(t) = 2, por el subárbol con raíz 1, hijo de 4. Si, a diferencia de este caso, no hubiera ningún SAPI en el árbol, se debería retornar 0.

- b) Explicar detalladamente lo mínimo qué se debería cambiar en el algoritmo anterior si, además de obtener el número del SAPI más grande, se deseara obtener la cantidad de veces que aparece, sin cambiar las cotas.

Ej. 3. Elección de Estructuras

Convocaron al equipo de Algo2 para colaborar en la creación de una nueva red social de mensajes cortos, llamada *Hornero*, que buscará reemplazar a otra que cayó en desgracia luego de un cambio poco afortunado de dueño.

En *Hornero* cada usuario tiene un identificador alfanumérico, de longitud máxima 20 caracteres. Los usuarios podrán publicar *posts* de un tamaño ilimitado, donde cada uno de ellos queda definido por una serie de palabras de tamaño acotado y puede contener menciones a otros usuarios (lo que se denota con una arroba antes del nombre del usuario). Por ejemplo, “Hoy jugamos al fútbol con mis amigos @EltonAlmizcle y @MrFeiknius” tiene diez palabras y hace mención a dos usuarios. Así, entre otras cosas, se quiere saber en qué mensajes se mencionó a algún usuario particular y obtener rápidamente todos los mensajes de un usuario. Siguiendo los preceptos iniciales de la vieja red social, no se podrán modificar mensajes una vez publicados, aunque estos sí podrán borrarse, con la consiguiente actualización del registro de menciones, entre otras cosas.

Dadas las siguientes operaciones, que son sólo algunas de las que posee el módulo, y de acuerdo a las complejidades temporales de peor caso indicadas, donde l es la cantidad de mensajes actuales del usuario, y p y m son la cantidad de palabras y la cantidad de menciones en un mensaje, respectivamente:

- **POSTEAR**(**inout** sistema: hornero, **in** idUsuario: id, **in** mensaje: secuencia(string)) \rightarrow idMensaje:int
 {Pre: El usuario idUsuario ya existe en el sistema.}
 Publica un nuevo mensaje de un usuario determinado, y devuelve el id del mensaje creado, que sólo es único para un mismo usuario (i.e., un mensaje de otro usuario puede tener el mismo id).
Complejidad: $O(\log(l) + p + p * m)$
 - **BORRAR**(**inout** sistema: hornero, **in** idUsuario: id, **in** idMensaje: int)
 {Pre: El idUsuario y el idMensaje ya existen en el sistema.}
 Borrar un mensaje determinado de un usuario.
Complejidad: $O(\log(l) + m)$
 - **VERMENCIONES**(**in** sistema: hornero, **in** idUsuario: id) \rightarrow menciones:secuencia(secuencia(string))
 {Pre: El usuario idUsuario ya existe en el sistema.}
 Obtiene todos los mensajes en donde se menciona al usuario.
Complejidad: $O(1)$
 - **VERTODOS**(**in** sistema: hornero, **in** idUsuario: id) \rightarrow mensajes:secuencia(secuencia(string))
 {Pre: El usuario idUsuario ya existe en el sistema.}
 Obtiene todos los mensajes del usuario.
Complejidad: $O(1)$
- a) Plantear la estructura de representación del módulo HORNERO, que provea las operaciones mencionadas más arriba. Se debe explicar qué información se guarda en cada parte, las relaciones entre ellas, y cómo afectarían al invariante de representación.
- b) Resumir de qué manera se resolvería cada una de las operaciones listadas anteriormente, de acuerdo la estructura elegida y la cota solicitada, y haciendo las aclaraciones necesarias sobre *aliasing*.
- c) Escribir el algoritmo de las operaciones POSTEAR y BORRAR y justificar de manera formal y detallada su complejidad de peor y mejor caso. Se puede suponer que existe una operación ESMENCIÓN? que dada una palabra nos dice si es una mención (i.e., el texto comienza con una '@') en $O(1)$.
- d) Realizar los cambios necesarios en estructura y algoritmos sin modificar las cotas requeridas previamente para poder agregar la siguiente operación:
- **VERRECIENTES**(**in** sistema: hornero, **in** idUsuario: id) \rightarrow mensajes:secuencia(secuencia(string))
 {Pre: El usuario idUsuario ya existe en el sistema.}
 Obtiene los r mensajes más recientes del usuario. Si hubiera menos que r , devuelve los que haya.
Complejidad: $O(1)$

Para la resolución del ejercicio no esta permitido utilizar módulos implementados con tabla hash de base.